Length Limited Coding and Optimal Height-Limited Binary Trees

Lawrence L. Larmore

University of California, Irvine

Technical Report 88-01

January 1988

Length Limited Coding and Optimal Height-limited Binary Trees

Lawrence L. Larmore[1] CSU Dominguez Hills and UC Irvine

Abstract

A new $O(nL)$-time algorithm is given for finding an optimal prefix-free binary code for a weighted alphabet of size n, with the restriction that no code string be longer than L. An $O(nL\log n)$-time algorithm is given for the corresponding alphabetic problem, which is equivalent to optimizing a dictionary of n words, implemented as a binary tree of height $h \leq L$ with all data in the leaves.

1. Introduction

The alphabetic coding problem. In [HuTu], Hu and Tucker present a solution to the following problem: given a list $\phi_1, \ldots \phi_n$ of non-negative weights ("frequencies") find a binary tree[2] T of size n (*i.e.*, with n leaves) which minimizes the *weighted depth* $\overline{T} = \sum_{i=1}^{n} \phi_i l_i$, where l_i is the depth of the i^{th} leaf of T. If n items must be stored in the leaves of a binary search tree T, and if ϕ_i is the frequency of access of the i^{th} item, then the expected time to find an item is proportional to $\overline{T}$. A tree which minimizes $\overline{T}$ is then an optimal binary search tree in this sense.

A very different application is optimizing alphabetic codes. Suppose Σ is an alphabet of size n, and ϕ_i is the frequency with which the i^{th} symbol of Σ is transmitted. A binary tree with n leaves then determines a prefix-free[3] binary code for Σ, and a tree which minimizes $\overline{T}$ determines a code where the expected length of a code string is minimized. Figure 1 shows the correspondence between a binary tree and a binary code.

Huffman's problem. If there is no requirement that the code be alphabetic, Huffman's algorithm finds an optimal code [Huf]. This non-alphabetic problem can be shown to be reduced to the alphabetic problem by simply sorting the weights [HuTa].

Height-limited optimal trees. Given an integer $L \geq \log_2 n$ and frequencies $\phi_1, \ldots \phi_n$, the *height-limited alphabetic optimal tree* problem is to find a tree T which minimizes $\overline{T}$, subject to the restriction that $l_i \leq L$ for all i. The *height limited optimal tree* problem is the same problem, where the weights may be arbitrarily permuted. (Without loss of generality, this permutation sorts them.) These two problems are sometimes referred to as the *length-limited coding problem* and the *length-limited alphabetic coding* problem.

Previous results. The (non-alphabetic) optimal coding problem is solved in $O(n\log n)$ time by Huffman's original algorithm [Huf]. The restricted length version was

---

[1] Author's address: Department of Information and Computer Science, University of California, Irvine, CA 92717. Net address: larmore@ics.uci.edu

[2] In this paper, a binary tree must be non-empty, and every non-leaf node must have exactly two children.

[3] A code is *prefix-free* if no code string is a prefix of any other.

solved in $O(nL2^L)$ time by Hu and Tan [HuTa], in $O(n^2L)$ time by Garey [Ga], and in $O(n^{1.5}L\log^{0.5}n)$ time by the author [L]. More recently, an $O(nL)$-time algorithm was found [LH]. This paper contains a considerably simplified $O(nL)$-time algorithm, which we call the *package-merge* algorithm.

The alphabetic coding problem can be solved in $O(n\log n)$ time by the Hu-Tucker and Garsia-Wachs algorithms [HuTu] [GaWa]. The restricted-length version is solved in $O(n^3L)$ time by Garey [Ga], in $O(n^2L)$ time by Itai and Wessler, independently [I] [W]. In this paper a new $O(nL\log n)$-time algorithm which we call the *alphabetic package-merge* algorithm is presented.

2. The package-merge algorithm

In this section, we introduce the *binary knapsack problem*, and the *package-merge* algorithm which solves the binary knapsack problem in essentially linear time. We then show how an instance of the length-limited coding problem with parameters n, L can be reduced to an instance of the binary knapsack problem of size nL. The package-merge algorithm thus solves the length-limited coding problem in $O(nL)$ time.

The binary knapsack problem. Given m items, where each has a *weight* and a *width*, both real numbers, we can ask for that set S of items of minimum total weight subject to the restriction that the total width of its members equals a given value Q. In general, this problem is *NP*-complete. We define the *binary knapsack* problem to be the special case where every width is of the form 2^{-d} for some $d \in N$.

The package-merge algorithm. We will assume that Q is an integer. The package-merge algorithm maintains lists of "packages." Each package is a set of items of total width 2^{-d} for some $d \in N$, and each list consists of packages of all the same width, sorted in order of increasing weight. Initially, each item is a package by itself, and each list is the list of all items of a given width. Each step of the algorithm combines the two smallest weight items of the smallest remaining width to form a single package of the next larger width, which is then inserted into the appropriate list. An odd package of width less than 1 is discarded. Finally, there is only one list, consisting of packages of width 1, sorted by weight. S is then taken to be the union of the first Q of these. (See figure 2.)

```
PACKAGE-MERGE ALGORITHM
Let D be such that 2^-D is the smallest width of any item
For each d, let A_d be the list of items of width 2^-d, sorted by weight
for d ← D downto 1 loop
        if A_d has odd length then discard its heaviest item
        Combine adjacent pairs of A_d to form a list B_d of packages of width 2^-d+1
        Merge B_d into A_d-1
end loop
let S be the union of the Q least weight items of A_0
```

The package-merge algorithm takes $O(m\log m)$ time if there are m items. However, if the items are pre-sorted (by width, then by weight within each width) the algorithm takes linear time. The algorithm can be modified to cover the case where Q is not an integer.

The reduction. Let $\phi_1, \ldots \phi_n$ be the list of frequencies (sorted into non-increasing order) for an instance of the (non-alphabetic) restricted length optimal coding problem, and let L be maximum permitted length. We define a *node* to be an ordered pair $(i,l) \in [1,n]\times[1,L]$. We define the *weight* of that node to be ϕ_i, and its *width* to be 2^{-l}. If T is any tree, define $nodeset(T) = \{(i,l) \mid 1 \le l \le l_i\}$ where l_i is the depth of the i^{th} leaf of T. Thus $T = \sum_{i=1}^{n} \phi_i l_i$ is exactly the total weight of $nodeset(T)$, and it can be shown (by induction on n) that the total width of $nodeset(T)$ is $n - 1$.

The reduction to the knapsack problem is to let each node be an item. Use the package-merge algorithm to find a set $A \subseteq [1,n]\times[1,L]$ of width $n-1$ of minimum weight. If ties (for equal weight nodes) are broken in favor of nodes of smaller index, A will be the nodeset of an optimal height-restricted tree. (See figure 3.)

Linear space. The package-merge algorithm for the restricted length coding problem can be run in linear space, using a trick similar to that used in [Hi]. The effect is to multiply time by a constant factor.

3. The Alphabetic package-merge algorithm

In this section, we assume given weights $\phi_1, \ldots \phi_n$, not necessarily sorted, and an integer $L \ge \log_2 n$. We give an $O(nL\log n)$-time algorithm to find an optimal height-limited alphabetic tree, *i.e.*, a binary tree T for which $T = \sum_{i=1}^{n} \phi_i l_i$ is minimized, where l_i is the depth of the i^{th} leaf of T.

Like the non-alphabetic version, this algorithm successively builds optimal packages at higher and higher level to construct the nodeset of an optimal tree. Each optimal package at level d is formed by combining two optimal packages at level $d+1$. As in the Hu-Tucker algorithm, packages can only be combined if they form a "compatible pair," defined below.

As in the previous section, we say that $(i,l) \in [1,n]\times[1,L]$ is a *node* of *index* i, of *weight* ϕ_i, of *level* l and of *width* 2^{-l}.

Forests. For $0 \le d \le L$ and $1 \le m \le n$, we define an *m-forest at level d* (abbreviated (d,m)-forest) to be a list of m trees, having n leaves altogether, where the root of each tree is taken to be at level d, and each leaf is at level no more than L. A (d,m)-forest F is characterized by the list $(l_1, \ldots l_n)$ of levels of its leaves (for example (2,3,4,4,2,2,3,3) represents the (2,5)-forest illustrated in figure 4) and we define $nodeset(F) = \{(i,l) \mid 1 \le l \le l_i\}$. We remark that a (d,m)-forest exists only if $m \ge 2^{d-L}n$, since each of the trees can have at most 2^{L-d} leaves.

Optimal packages. Define $Best\_Forest[d,m]$ to be that (d,m)-forest for which $T =$

$\sum_{i=1}^{n} \phi_i l_i$ is minimized, and let $N[d,m] = Nodeset(Best\_Forest[d,m])$. Note that $T = weight(N[d,m])$. (In case of candidates of equal weight, ties must be broken by a systematic left-first rule.) Define $P[d,k] = N[d,n-k] - N[d,n-k+1]$, which we call the k^{th} *optimal combined d-package*, defined for all $1 \leq k \leq \lfloor(1-2^{d-L})n\rfloor$. We define an *optimal d-package* to be either a combined optimal d-package, or a singleton $\{(i,d)\}$. Figure 5 illustrates all optimal combined packages for a small example.

The following theorems are stated without proof:

Theorem 1: $N[d,k] \subseteq N[d,k+1]$.
Corollary: $width(P[d,k]) = 2^{-d}$.
Theorem 2: $P[d,k]$ is the disjoint union of two optimal $(d+1)$-packages.

Best_Forest[0,1] is clearly the optimal tree, the desired output of the algorithm. Its nodeset is the union $P[0,1] \cup ... \cup P[0,n-1]$. The alphabetic package-merge algorithm successively constructs, for decreasing d, the optimal combined packages at level d. In high-level form, here is the algorithm:

THE ALPHABETIC PACKAGE-MERGE ALGORITHM
for $d \leftarrow L-1$ **downto** 0 **loop**
    for $k \leftarrow 1$ **to** $\lfloor(1-2^{d-L})n\rfloor$ **loop**
        $P[d,k] \leftarrow$ union of the least weight *compatible pair* of remaining optimal $(d+1)$-packages
    end loop
end loop
$T^{opt} \leftarrow$ that tree whose nodeset is $P[0,1] \cup ... \cup P[0,n-1]$

Compatible pairs. We say that an optimal $(d+1)$-package is "remaining" if it was not incorporated into any of $P[d,1]$, ... $P[d,k-1]$. The remaining optimal $(d+1)$-packages are of two kinds: singletons and optimal combined packages. Each is assigned an *index*, that being the greatest index of any of its member nodes. Two remaining packages are *incompatible* if they have indices i and j and if there is a remaining singleton whose index lies strictly between i and j. Otherwise, they form a *compatible pair*. Correctness of the package-merge algorithm depends on the following theorem, given without proof here:

Theorem 3: $P[d,k]$ is the union of the least weight pair of compatible unused optimal $(d+1)$-packages.

Implementation. The alphabetic package-merge algorithm can be implemented to run in $O(n\log n)$ time for each level d (and hence in $O(nL\log n)$ time altogether) using "mergeable priority queues." (See [AHU] for example.) If i_1, ... i_m are the indices of the remaining singletons, let C_r be the set of remaining optimal $(d+1)$-packages whose

indices lie in the interval $[i_r, i_{r+1}]$. (Let $i_0 = 0$, $i_{m+1} = n+1$.) Then two remaining packages are compatible if and only if they both lie in some C_r. A remaining package can belong to either one or two of the C_r. When the least weight compatible pair is found, the one or both copies of each $(d+1)$-package must be removed from the structure, and possibly some adjacent sets must be merged.

In figure 6, we show a hand implementation of the alphabetic package-merge algorithm.

References

[AHU] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).

[GaWa] A.M. Garsia and M.L. Wachs, A New algorithm for minimal binary search trees, *SIAM J Comp* **6** (1977) pp. 622-642.

[Ga] M.R. Garey, Optimal Binary Search Trees with Restricted Maximal Depth, *SIAM J Comp* **3** (1974) pp. 101-110.

[Hi] Hirschberg, D.S., A linear space algorithm for computing maximal common subsequences, *Comm ACM 18* **6** (1975), pp. 341-343.

[HuTa] T.C. Hu and K.C. Tan, Path length of binary search trees, *SIAM J Applied Math* **22** (1972) pp. 225-234.

[HuTu] T.C. Hu and A.C. Tucker, Optimal computer search trees and variable length alphabetic codes, *SIAM J Applied Math* **21** (1971) pp. 514-532.

[Hu] T.C. Hu, *Combinatorial Algorithms*, Addison Wesley (1982).

[Huf] D.A. Huffman, A Method for the construction of minimum redundancy codes, *Proc. Inst. Radio Engineers* **40** (1952) pp. 1098-1101.

[I] Itai, Alon, Optimal alphabetic trees, *SIAM Journal of Computing* **5**(1976), pp. 9-18

[L] L.L. Larmore, Height-restricted optimal binary trees, *SIAM Jour. on Comp.*, to appear, December 1987.

[LH] L.L. Larmore and D.S. Hirschberg, A Fast Algorithm for Optimal Length-Limited Codes, submitted for publication.

[W] Wessner, Rusell L., Optimal alphabetic search trees with restricted maximal height, *Information Processing Letters* **4** (1976), pp. 90-94

Figure 1. The binary tree corresponding to a prefix-free binary code.

a	$\rightarrow$	0
b	$\rightarrow$	101
c	$\rightarrow$	11
d	$\rightarrow$	100

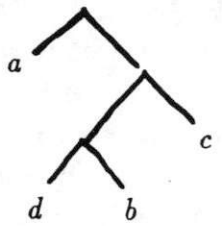


Figure 2. The package-merge algorithm. Each "rectangular" node is an original item, also a singleton package. Each "round" node is a combined package. The width of a package at level d is 2^{-d}. The set S of width Q ($Q = 3$ in this case) of minimum weight is represented by the square nodes enclosed by the curve. The algorithm is worked from the bottom of the diagram up.

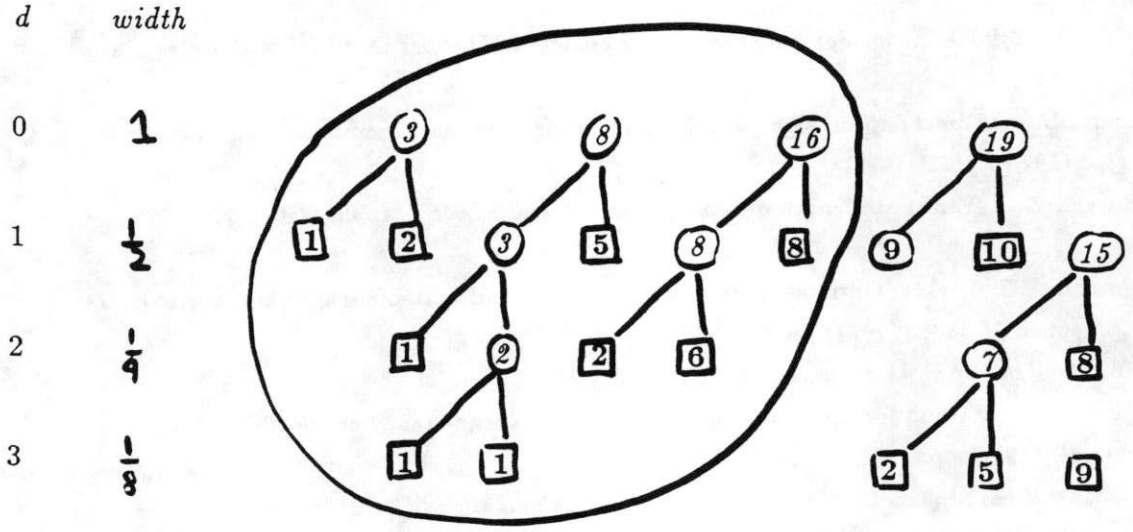


Figure 3. The minimum weight nodeset of width $n-1$ and the resulting optimal tree. Each node is shown as a number which is its weight, ϕ_i. The width of each node in the d^{th} row is 2^{-d}. ($L = 4$, $n = 7$, and the frequencies are 1, 1, 2, 5, 8, 20, 25.)

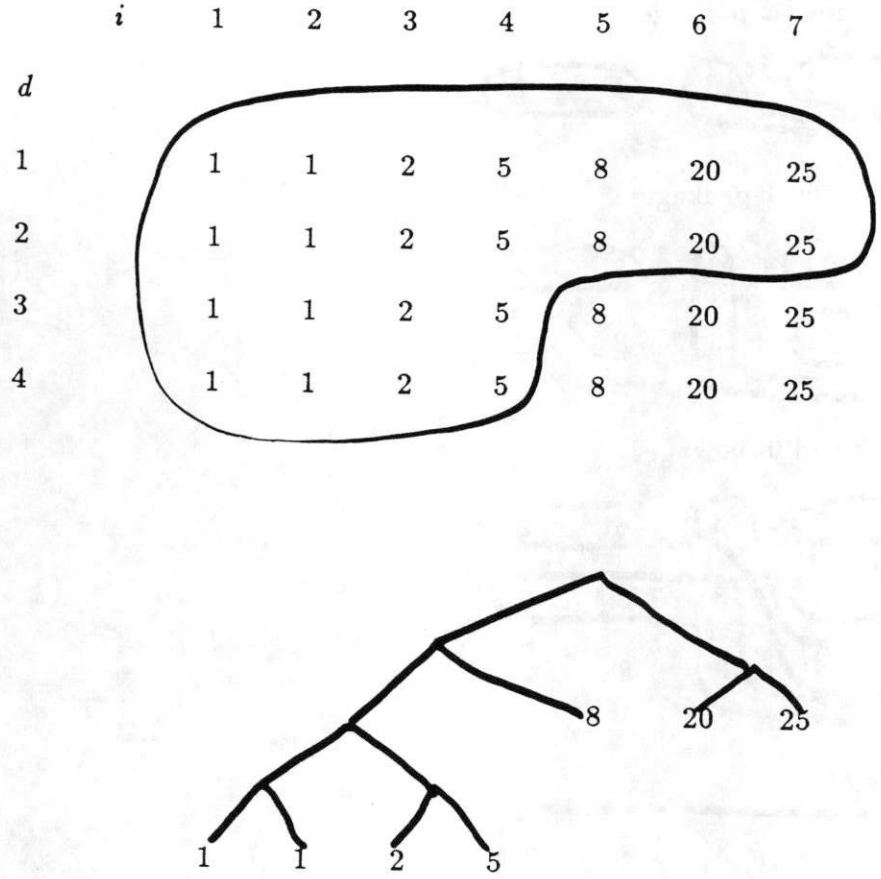


Figure 4. A (2,5)-forest and its nodeset. Each node is represented by a dot.

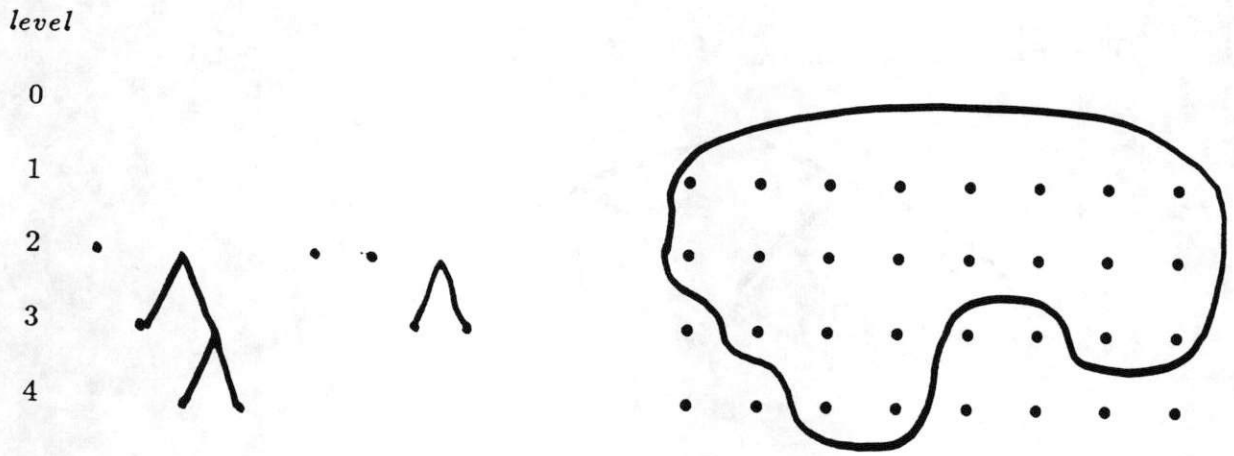


Figure 5. Optimal combined packages at each level. $L = 3$, and the list of frequencies is 2, 1, 1, 3, 8, 2.

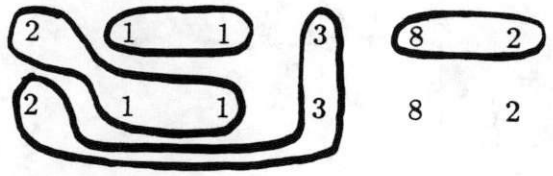
level The optimal combined 2-packages

3

level The optimal combined 1-packages

2

3

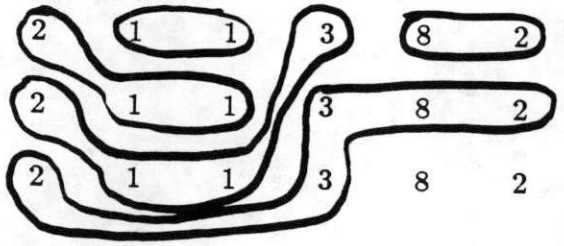


level The optimal combined 0-packages

1

2

3

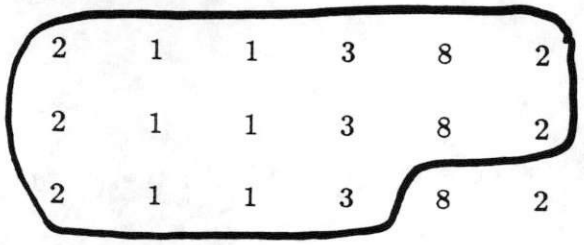


level The optimal nodeset

1

2

3



The optimal alphabetic tree

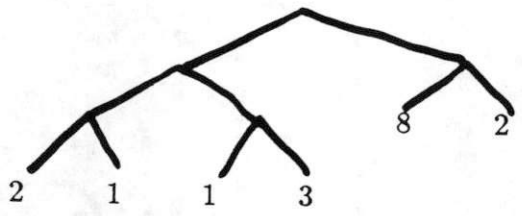
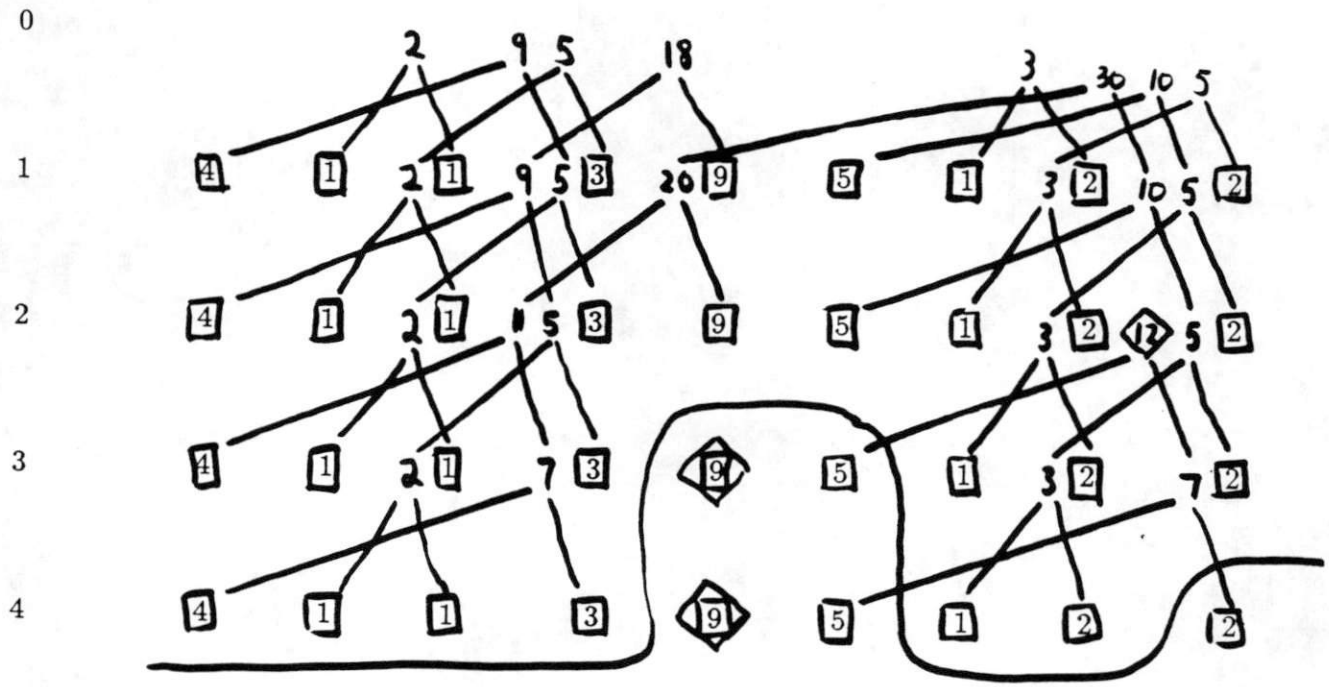


Figure 6. Hand implementation of the alphabetic package-merge algorithm. Work proceeds from the bottom of the figure up. The nodes are written in a rectangular array. Optimal combined packages are formed and merged into the appropriate position on the next higher row. The final structure is a forest of $n-1$ trees rooted at level 0, as well as up to L "orphan" trees rooted at various levels. The optimal nodeset consists of the leaves of the n-1 trees at level 0. In the example shown, $L = 4$. Roots of the orphan trees are marked as "diamond" packages. The nodeset of the optimal tree consists of all the nodes above the curve.



The optimal tree